Apuntes U 1

Tema ***Análisis de Algoritmos***

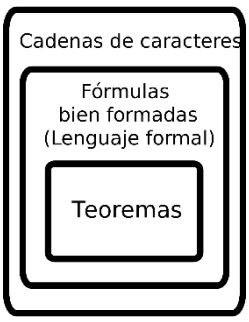
"Los programadores malos se preocupan por el código.
Los buenos programadores se preocupan por las estructuras de datos y sus relaciones."
Linus Torvalds

¿Qué es un Lenguaje de Programación?

Un lenguaje de programación es un lenguaje formal diseñado para expresar procesos que pueden ser llevados a cabo por máquinas como las computadoras.
En matemáticas, lógica, y ciencias de la computación, un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados.

¿Qué es un Lenguaje Formal?

En matemáticas, lógica y ciencias de la computación, un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados. Al conjunto de los símbolos primitivos se le llama el alfabeto (o vocabulario) del lenguaje, y al conjunto de las reglas se lo llama la gramática formal (o sintaxis). A una cadena de símbolos formada de acuerdo a la gramática se la llama una fórmula bien formada (o palabra) del lenguaje. Estrictamente hablando, un lenguaje formal es idéntico al conjunto de todas sus fórmulas bien formadas. A diferencia de lo que ocurre con el alfabeto (que debe ser un conjunto finito) y con cada fórmula bien formada (que debe tener una longitud también finita), un lenguaje formal puede estar compuesto por un número infinito de fórmulas bien formadas.



¿Qué son los paradigmas de Programación?

Un paradigma de programación es una propuesta tecnológica adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que únicamente trata de resolver uno o varios problemas claramente delimitados.
En general, la mayoría de paradigmas son variantes de los dos tipos principales de programación, imperativa y declarativa. En la programación imperativa se describe paso a paso un conjunto de

instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

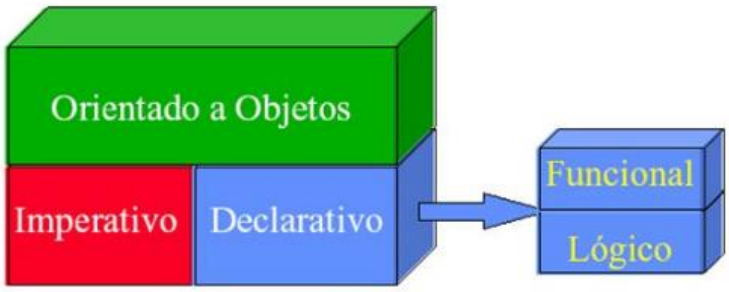
Imperativo o por procedimientos: es considerado el más común y está representado es decir elogiado, por ejemplo, por C, BASIC o Pascal.
Declarativo: por ejemplo la programación funcional, la programación lógica, o la combinación lógico-funcional.

a. Funcional: está representado por Scheme o Haskell. Este es un caso del paradigma declarativo.
b. Lógico: está representado por Prolog. Este es otro caso del paradigma declarativo.

Orientado a objetos: está representado por Smalltalk, un lenguaje completamente orientado a objetos.

Programación dinámica: está definida como el proceso de romper problemas en partes pequeñas para analizarlos.

Programación multiparadigma: es el uso de dos o más paradigmas dentro de un programa.



¿Qué es un Algoritmo?

El término algoritmo proviene del árabe **al-Khowârizmî**, sobrenombre del célebre matemático árabe **Mohámed ben Musa**. En el diccionario de la Real Academia española se define como el conjunto ordenado y finito de operaciones que permiten hallar la solución a un problema. El término de algoritmo se puede entender como la descripción de cómo resolver un problema. El conjunto de instrucciones que especifican la secuencia de operaciones a realizar, en orden, para resolver un sistema específico o clase de problemas, también se denomina algoritmo. En otras palabras un algoritmo es una "especie de fórmula" para la resolución de un problema.

Propiedades:

1) **Ser finito**. Un algoritmo tiene que acabar tras un número finito de pasos. El número de pasos de un algoritmo, por grande y complicado que sea el problema que soluciona, debe ser limitado. Todo algoritmo incluye los pasos inicio y fin.

2) **Precisión**. Esto significa que los pasos u operaciones del algoritmo deben desarrollarse en un orden estricto.

3) **Ser definido**. Cada paso de un algoritmo debe de tener un significado preciso; las acciones a realizar han de estar especificadas en cada paso rigurosamente y sin ambigüedad.

El algoritmo se desarrolla como paso fundamental para desarrollar un programa, el computador sólo desarrollará las tareas programadas, con los datos suministrados; no puede improvisar ni inventa el dato que necesite para realizar un paso. Por ello, cuantas veces se ejecute el algoritmo, el resultado depende estrictamente de los datos suministrados, de hecho si se ejecuta con un mismo conjunto de datos de entrada, el resultado será siempre el mismo.

4) **Presentación formal**: el algoritmo debe estar expresado en alguna de las formas comúnmente aceptadas. Las formas de presentación de algoritmos son, entre otras: el pseudocódigo, diagrama de flujo y diagramas de Nassi/Schneiderman.

5) **Conjunto de entradas**. Debe existir un conjunto específico de objetos, cada uno de los cuales constituye los datos iniciales de un caso particular del problema que resuelve el algoritmo. A este conjunto se llama conjunto de entrada del algoritmo.

6) **Conjunto de salidas**. Debe existir un conjunto específico de objetos, cada uno de los cuales constituye la salida o respuesta que debe tener el algoritmo para los diferentes casos particulares del problema. Para cada entrada del algoritmo debe existir una salida asociada.

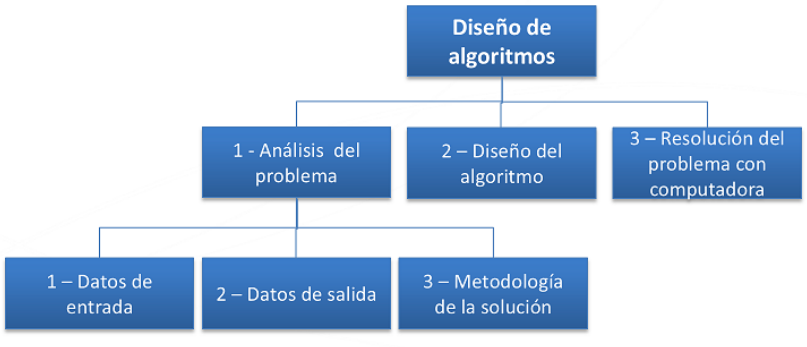
7) **Efectividad (o Corrección)**. El algoritmo debe satisfacer la necesidad o solucionar el problema para el cual fue diseñado. Para garantizar que el algoritmo logre el objetivo, es necesario ponerlo a prueba; a esto se le llama verificación o prueba de escritorio.

8) **Eficiencia**: En cuanto menos recursos requiere será más eficiente el algoritmo. Este concepto se aplica, entre otros puntos, las formas de almacenar los datos, de leerlos, etc.

Datos e información: En el uso diario, datos e información son esencialmente sinónimos. Sin embargo, en las ciencias de la computación suelen hacer una diferencia: datos se refiere a la representación de algún hecho, concepto o entidad real (los datos pueden tomar diferentes formas, por ejemplo, palabras escritas o habladas, números y dibujos); información que implica datos procesados y organizados

Diseño de algoritmos

Los métodos más eficaces para el proceso de diseño se basan en el conocido **divide y vencerás**, es decir, la resolución de un problema complejo se realiza dividiendo el problema en sub-problemas y a continuación dividir estos sub-problemas en otros de nivel más bajo o sea más simples, hasta que pueda ser implementada una solución en la computadora y así sucesivamente (de lo global a lo concreto); es lo que se denomina diseño descendente **Top-Down design o modular**.



Una vez realizado un primer acercamiento al problema, éste se ha de ampliar, (romper el problema en cada etapa y expresar cada paso en forma más detallada) lo que denominamos como **refinamiento del algoritmo o Stepwise Refinement**.

Cada sub-problema es resuelto mediante un **módulo o sub-programa**, que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un programa principal (el módulo de nivel más alto) que llama a sub-programas (módulos de nivel más bajo) que a su vez puede llamar a otros sub-programas.
Los programas estructurados de esta forma se dicen que tienen un diseño modular y el método de romper el programa en módulos más pequeños se llama programación modular.
Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre ellos.
El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar el módulo.
2. Comprobar el módulo.
3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular, con refinamiento sucesivos que permitan una posterior traducción a un lenguaje, se denomina diseño del algoritmo.
En todos los casos, durante el desarrollo de la materia, evitaremos focalizarnos en un lenguaje y de igual manera todo ejemplo o ejercicio que les pida deberá ser resuelto con las instrucciones propias, que veamos dentro de algoritmos. Por ejemplo: Resumiendo todo lo visto hasta ahora, en diseño top-down, modular sería:

Es decir que resolver un problema, implica la resolución de tres sub-problemas en el siguiente orden:

1º. Análisis del problema
2º Diseñar el algoritmo
3º Aplicar la solución en la computadora.

Obviamente cada uno de estos sub-problemas pueden a su vez dividirse en otros; por ejemplo:

Con lo cual vemos que el concepto de dividir un gran problema en pequeños problemas, nos ayuda a:

- Reducir la complejidad para el desarrollo
- Facilitar la comprensión de la lógica de cada proceso
  Cada proceso pasa a tener un objetivo único
- Cualquier cambio se aplica sobre el proceso específico que maneja la situación afectada
- Permite reducir los futuros costos de mantenimiento.
- Permite entregas parciales que ayudan a reducir los tiempos totales.

Dato, Campo, Campo Estructura

Podemos dar como primera definición, que un CAMPO es el elemento que nosotros identificamos mediante un nombre, por lo cual una de las características fundamentales del CAMPO es su nombre; ese nombre que declaramos es el que permite que el algoritmo lo identifique unívocamente y que mediante el mismo lo utilizare en el proceso en las operaciones que desarrollemos. Mientras que el DATO es el valor que ese campo tiene.

Ese valor lo puede obtener desde:

- El exterior

• Por modificaciones durante el proceso.

Por tal motivo consideramos al CAMPO como el nombre y lugar físico en el cual alojaremos el valor o DATO.

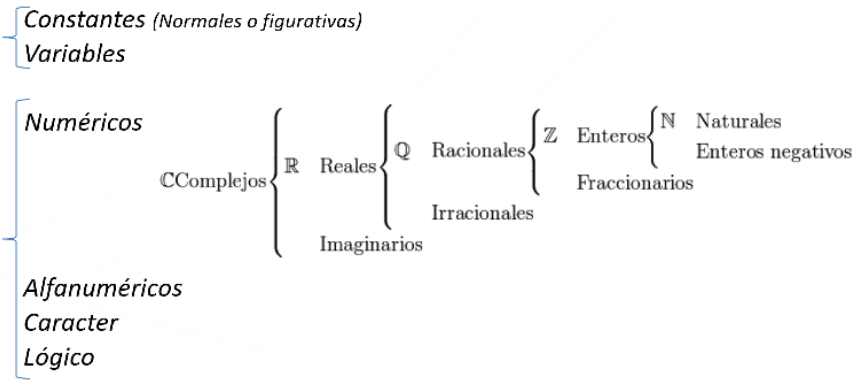
Constante (definición): Un objeto de tipo constante es un valor cuyo contenido no puede variar.
Podríamos diferenciar:

• Constante Normal: Valor constante expresado en si mismo. Por Ej.: Valor numérico 128
• Constante figurativas: Un nombre que de manera figurada simboliza un valor constante que no cambia. Por Ej.: PI = 3,1416; en el que PI sería la constante.

Variable (definición): Lugar en la memoria que está reservado, y que posee un nombre, un tipo y un contenido. El valor de la variable puede cambiar y puede ser modificado a lo largo del programa. La variable tiene un valor determinado, de forma que el tipo de variable, debe de ser uno en concreto, por lo tanto el primer dato de la variable condicionará los posteriores datos que almacenará después.

La variable contiene tres cualidades:
- Nombre
- Tipo
- Valor

Resumiendo
Podemos encontrar solamente 4 tipos de datos en nuestros ejercicios:
• Real
• Entero
• Alfanumérico
• Lógico



Estructura de Datos

Es el conjunto de variables agrupadas y organizadas de cierta forma para representar un información y/o un comportamiento.

En el desarrollo de programas, existe una fase previa a la escritura del programa, esta es el **diseño del algoritmo** que nos guiará a la solución del problema, en esta fase también deberá considerarse la/las estructura/s de datos que se utilizarán.

El término estructura de datos se refiere a la forma en que la información esta organizada dentro de un programa. La correcta organización de datos puede conducir a algoritmos más simples y más eficientes.

Clasificación según su forma de locación

* Estructuras de datos **estáticas**: Son aquellas cuyo tamaño en memoria es fijo, por ejemplo, los arreglos.

* Estructuras de datos **dinámicas**: Son las estructuras que permiten variar su tamaño en memoria de acuerdo a las necesidades del ambiente, por ejemplo, listas enlazadas.

Clasificación según su lugar de locación

* **Lineales**: Son aquellas que se alojan en espacios contiguos de memoria.

* **No Lineales**: Son las estructuras que permiten ser alojadas en espacios no contiguos de memoria.

Estructura de Datos numéricas

1 bit. Bit es el acrónimo Binary digit ('dígito binario'), representa 2 estados y sus valores posibles son 0 o 1

1 nibble = 4 bits. Puede representar todas las combinaciones posibles con 4 bits, o sea 16 valores posibles; de 0 a 16. Se utiliza normalmente para el sistema hexadecimal.

1 Byte = 8 bits. Representa 256 valores posibles, numéricamente del 0 al 255. También se utiliza normalmente para representar un carácter *(Ver código ASCII)*.

1 word = XXBits. En computadoras viejas de arquitectura de 16 bits, un Word era representado por 16 bits, en las actuales de arquitectura de 32 y/o 64 Bits un Word tiene 32 y 64 bits respectivamente. Esto está atado a la tecnología o arquitectura de la computadora ya que es la cantidad de Bits que puede direccionar. También existe el Dword (2 veces el tamaño de un Word)

Kilo	10^3	2^{10}
Mega	10^6	2^{20}
Giga	10^9	2^{30}
Tera	10^{12}	2^{40}
Peta	10^{15}	2^{50}
Exa	10^{18}	2^{60}
Zetta	10^{21}	2^{70}
Yotta	10^{24}	2^{80}
Xona	10^{27}	2^{90}
Weka	10^{30}	2^{100}

Tipos Primitivos. No poseen métodos, no son objetos y no necesitan ser invocados para ser creados.

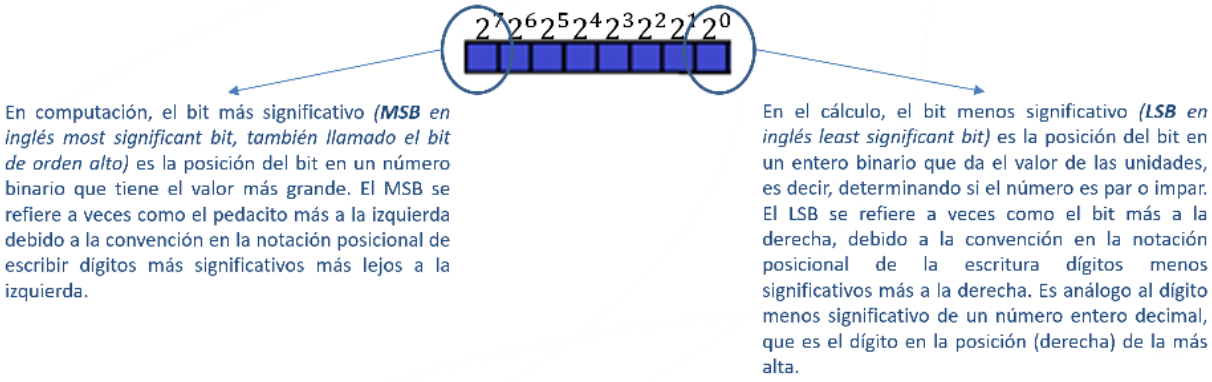
Tipos Objeto. Poseen métodos y necesitan ser invocados (instanciados) para ser creados.

Tipos Primitivos.

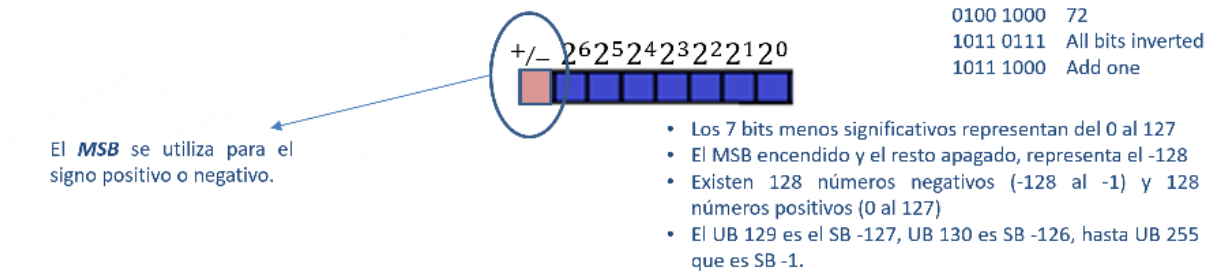
NOMBRE	TIPO	OCUPA	RANGO APROXIMADO
byte	Entero	1 byte	-128 a 127
short	Entero	2 bytes	-32768 a 32767
int	Entero	4 bytes	$2*10^9$
long	Entero	8 bytes	Muy grande
float	Decimal simple	4 bytes	Muy grande
double	Decimal doble	8 bytes	Muy grande
char	Carácter simple	2 bytes	---
boolean	Valor true o false	1 byte	---

Estructura “byte” (de 8 bits, 256 números representables)

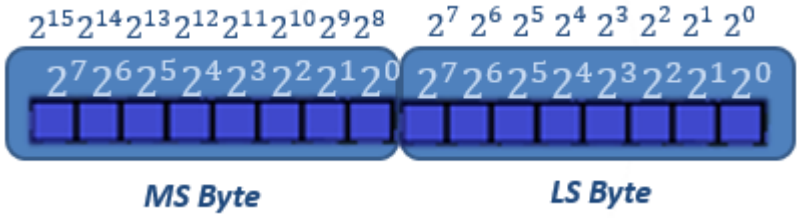
Byte *Unsigned (sin signo),(en c++* ***unsigned char****, en pascal* ***byte****)*:



Byte *Signed (con signo), (en Java Byte, en c++* ***signed char****, en pascal* ***shortint****)*:



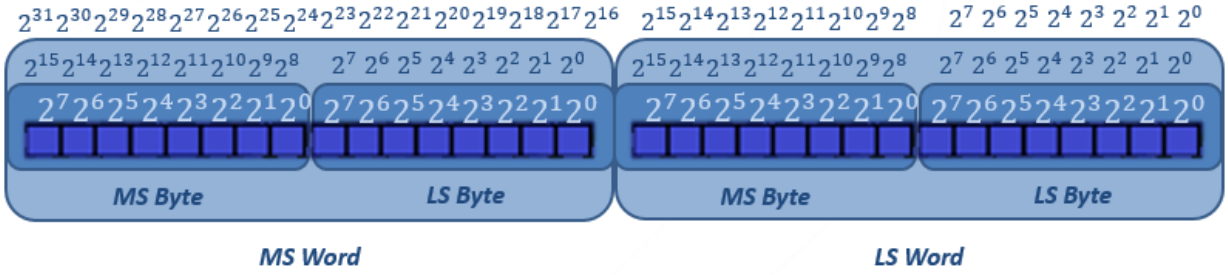
word *Unsigned (En Java no esta definido, en c++* ***unsigned int****, en Pascal* ***word****)*



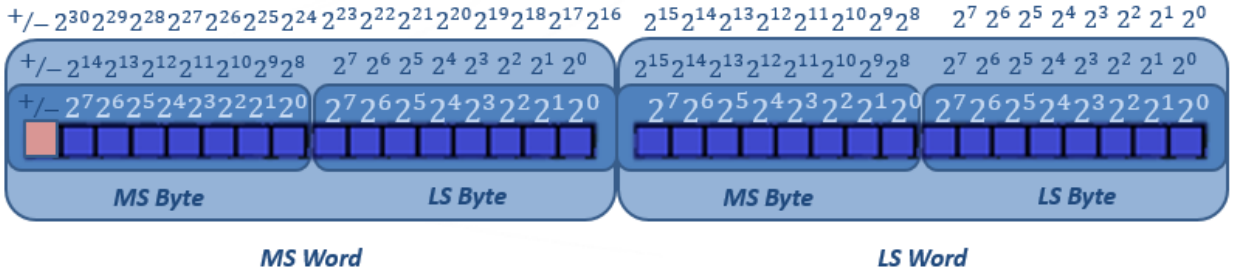
word ***Signed (En Java short, en c++ int, e Pascal integer)***

Estructura “longint o longword” (32 bits o 4 bytes, 4294967295 números representables)

longword *Unsigned (En Java no esta definido, en c++* ***unsigned long int****, en Pascal* ***longword****)*



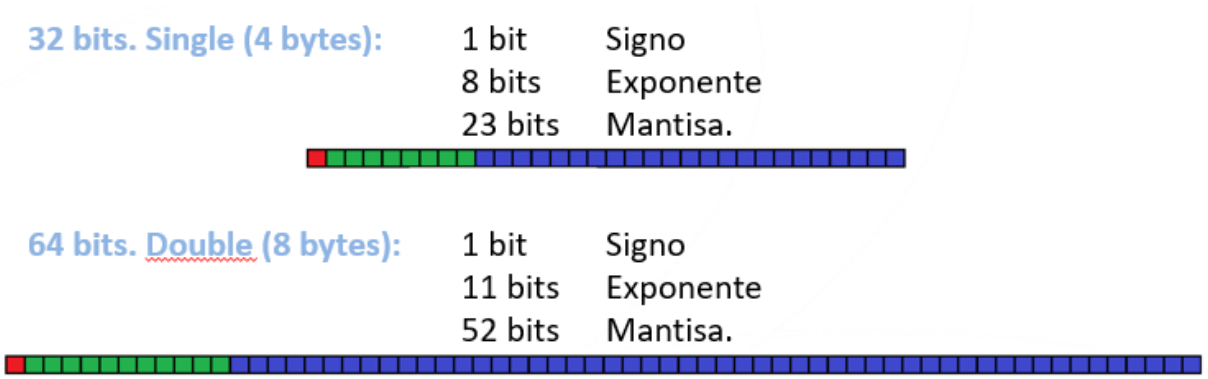
longint *Signed (En Java* ***int****, en c++* ***unsigned long int****, en Pascal* ***longword****)*



Estructura “float” o coma flotante

Norma IEEE 754, representación de coma flotante:

http://754r.ucbtest.org/standards/754xml.html
http://es.wikipedia.org/wiki/IEEE_coma_flotante



Ejemplo: expresar el número decimal -118,625 usando el sistema IEEE coma flotante.

1) Asignar al bit de signo el valor de "1", dado que el número decimal a convertir es negativo.
2) Transformar en binario -118,625 y queda 1110110,101 *(Se transforma la parte entera a binario. Luego se sigue con la parte fraccionaria, multiplicando cada número por 2. Si el resultado obtenido es mayor o igual a 1 se anota como un uno (1) binario. Si es menor que 1 se anota como un 0 binario. Ej., al multiplicar 0.6 por 2 obtenemos como resultado 1.2 lo cual indica que nuestro resultado es un uno (1) en binario, solo se toma la parte decimal del resultado).*
3) Mover la coma decimal a la izquierda, dejando sólo un 1 a su izquierda: 1110110,101=1,110110101*26. Esto es un número normalizado en coma flotante. El "1" anterior a la coma no se representa en la expresión resultante.
4) Asignar a la mantisa todos los bits a la derecha de la coma decimal, rellenado con ceros a la derecha hasta obtener los 23 bits que ocupa el campo. Es decir, 11011010100000000000000.
5) Sumar al exponente (6) el número 127, convirtiéndose en un exponente desplazado. El total resultante (133) se convierte en un número binario (10000101). El resultado se coloca en el campo "E".

Operaciones fundamentales en algoritmos computacionales.

- *Asignación.*
- *Operaciones Numéricas*
  - *Contadores*
  - *Acumuladores*
- *Operaciones Alfabéticas o Alfanuméricas*
  - *Len*
  - *Concatenación*
- *Operaciones de entrada-salida.*
  - *Entrada*
    - *Ingresar*
    - *Leer*
    - *Eof*
  - *Salida*

- *Mostrar*
- *Imprimir*
- *Escribir*

Representación de Algoritmos.

Pseudocódigo

En ciencias de la computación, y análisis numérico el pseudocódigo (o falso lenguaje) es una descripción de alto nivel compacta e informal1 del principio operativo de un programa informático u otro algoritmo.

Pseudocódigo estilo Pascal:

```
procedimiento bizzbuzz
para i := 1 hasta 100 hacer
    establecer print_number a verdadero;
    Si i es divisible por 3 entonces
        escribir "Bizz";
        establecer print_number a falso;
    Si i es divisible por 5 entonces
        escribir "Buzz";
        establecer print_number a falso;
    Si print_number, escribir i;
    escribir una nueva lina;
fin
```

Pseudocódigo estilo C:

```
subproceso funcion bizzbuzz
para (i <- 1; i<=100; i++) {
    establecer print_number a verdadero;
    Si i es divisible por 3
        escribir "Bizz";
        establecer print_number a falso;
    Si i es divisible por 5
        escribir "Buzz";
        establecer print_number a falso;
    Si print_number, escribir i;
    escribir una nueva línea;
}
```

Utiliza las convenciones estructurales de un lenguaje de programación real2 , pero está diseñado para la lectura humana en lugar de la lectura mediante máquina, y con independencia de cualquier otro lenguaje de programación. Normalmente, el pseudocódigo omite detalles que no son esenciales para la comprensión humana del algoritmo, tales como declaraciones de variables, código específico del sistema y algunas subrutinas. El lenguaje de programación se complementa, donde sea conveniente, con descripciones detalladas en lenguaje natural, o con notación matemática compacta. Se utiliza pseudocódigo pues este es más fácil de entender para las personas que el código del lenguaje de programación convencional, ya que es una descripción eficiente y con un entorno independiente de los principios fundamentales de un algoritmo. Se utiliza comúnmente en los libros de texto y publicaciones científicas que se documentan varios algoritmos, y también en la planificación del desarrollo de programas informáticos, para esbozar la estructura del programa antes de realizar la efectiva codificación.

No existe una sintaxis estándar para el pseudocódigo, aunque los ocho IDE's que manejan pseudocódigo tengan su sintaxis propia. Aunque sea parecido, el pseudocódigo no debe confundirse con los programas esqueleto que incluyen código ficticio, que pueden ser compilados sin errores. Los diagramas de flujo y UML pueden ser considerados como una alternativa gráfica al pseudocódigo, aunque sean más amplios en papel.

El pseudocódigo es una herramienta de programación en la que las instrucciones se escriben en palabras similares al español o inglés, que facilitan tanto la escritura como la lectura de programas, y permite a quien lo realiza concentrarse en las estructuras de control sin tener presente las características propias de algún lenguaje en especial. En esencia se puede definir como un lenguaje de especificaciones de algoritmos.
El pseudocódigo se encuentra a un solo paso de pasarse a un lenguaje de programación y es entendible, de igual forma que los diagramas de flujo, a diferencia que no es gráfico pero igual de comprensible.

Cuenta con palabras reservadas y la forma de su uso, y tiene una estructura mínima que se debe respetar. Siempre la primera palabra debe ser:
Algoritmo: nombre
Seguida de " : " y el nombre que le definamos para el mismo.
Debe tener una sentencia única de FIN, que indica la finalización del proceso.

¿Que es Indentación? Es un anglicismo (de la palabra inglesa *indentation*) de uso común en Informática y significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores para separarlo del texto adyacente, lo que en el ámbito de la imprenta se ha denominado siempre como *sangrado* o sangría.
En los lenguajes de Computación, la indentación se utiliza para mejorar la legibilidad del código por parte de los programadores, teniendo en cuenta que raramente se consideran los espacios en blanco como sentencias de un programa. Sin embargo, en ciertos lenguajes de programación como Haskell, Occa, y Python, la indentación se utiliza para delimitar la estructura del programa permitiendo establecer bloques de código.
Son frecuentes discusiones entre programadores sobre cómo o dónde usar la indentación, si es mejor usar espacios en blanco o tabuladores, ya que cada programador tiene su propio estilo. Vale aclarar que esta palabra no está reconocida por la Real Academia de Lengua, y en su lugar se debería usar SANGRADO, pero por el uso y costumbre que tiene en la jerga informática para nuestro curso la tomamos como válida, por lo que siempre debemos intentar identificar o marcar los bloques de instrucción de esta manera, lo cual ayuda a la legibilidad del código.

Diagramas de flujo

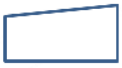
También llamado ordinograma, esta técnica muestra los algoritmos de una manera clara y comprensible.
Es una representación gráfica, es decir, se vale de diversos símbolos para representar las ideas o acciones a desarrollar.
Los diagramas de flujo ayudan en la comprensión de la operación de las estructuras de control. Por ejemplo: el inicio y el fin del algoritmo se representan con un símbolo elíptico, las entradas y salidas con un paralelogramo, las decisiones con un rombo, los procesos con un rectángulo, etc.
Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (ANSI) y algunos de ellos son

- *Terminador (inicio o fin)*

- *Conectores*

- *Definición de variables*
- *Asignación de valores a variables*
- *Declaración de expresiones Numéricas*

- *Asignaciones indirectas por teclado de variables en un diagrama de flujo*



- *Visualización por pantalla en un diagrama de flujo*

- *Sub-programa (sub rutina)*

- *Condición lógica*

- *Entrada / Salida*

- *Impresión*

- *Operación de cinta (secuencial)*

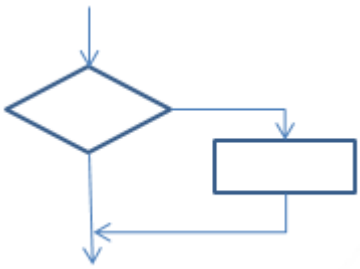
- *Operación de disco (random)*

- *Iteración (for, repetir y mientras)*

Estructuras en un diagrama de Flujo

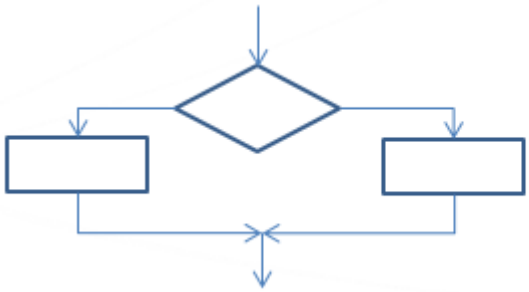
- ***Secuencial**: es la manera natural de organizar las acciones.*

- ***Selectiva simple***



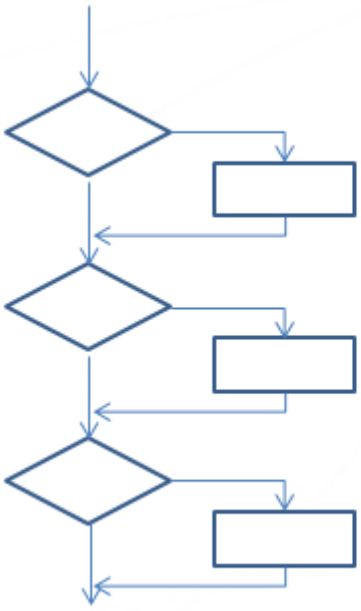
si [condición]
entonces
operación
sino
fin si

- ***Selectiva doble***



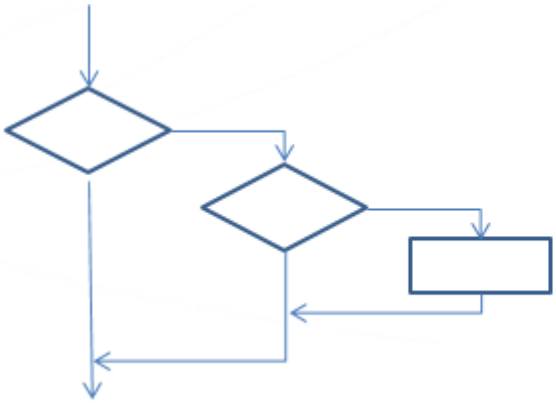
si [condición]
entonces
operación1
sino
operación2
fin si

- ***Selectiva múltiple***



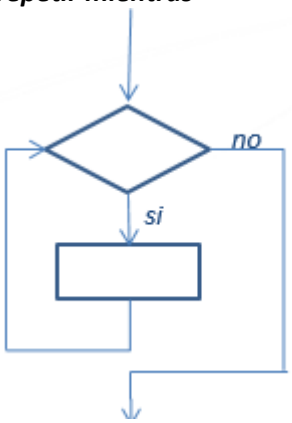
```
si [condición1]
entonces
                operación1
sino
fin si
si [condición2]
entonces
                operación2
sino
fin si
si [condición3]
entonces
                operación3
sino
fin si
```

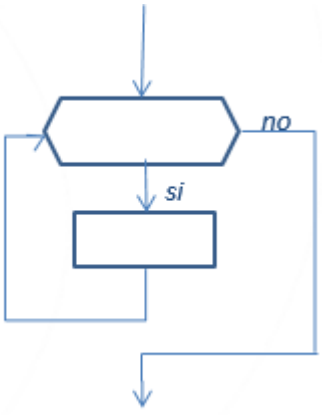
- ***Selectiva anidada***



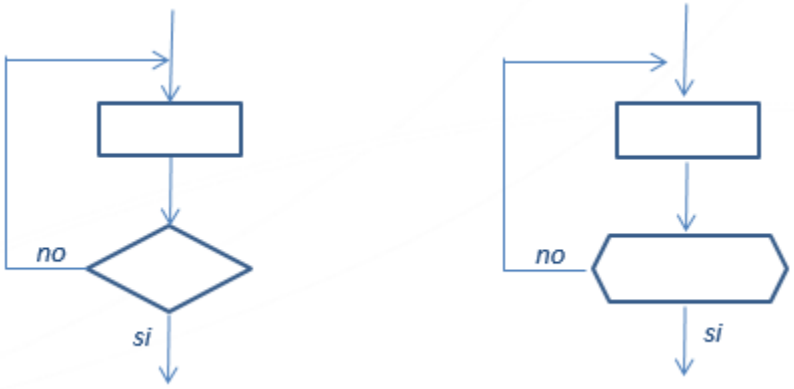
```
si [condición1]
entonces
                si [condición2]
                entonces
                                operación1
                sino
                fin si
sino
fin si
```

- ***Iterativa repetir mientras***





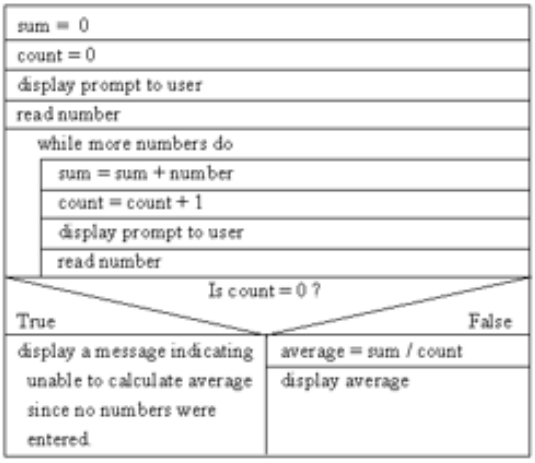
- ***Iterativa repetir hasta***



- ***Iterativa Desde – Hasta*** *(estructura igual al repetir mientras con condiciones distintas)*

Diagramas de Nassi-Shneiderman (o estructograma)

En programación de computadores un diagrama Nassi-Shneiderman (o NSD por sus siglas en inglés), también conocido como diagrama de Chapin1 2 es una representación gráfica que muestra el diseño de un programa estructurado.



Fue desarrollado en 1972 por Isaac Nassi y Ben Shneiderman. Este diagrama también es conocido como estructograma, ya que sirve para representar la estructura de los programas. Combina la descripción textual del pseudocódigo con la representación gráfica del diagrama de flujo.
Basado en un diseño top-down (de lo complejo a lo simple), el problema que se debe resolver se divide en subproblemas cada vez más pequeños - y simples - hasta que solo queden instrucciones simples y construcciones para el control de flujo. El diagrama Nassi-Shneiderman refleja la descomposición del problema en una forma simple usando cajas anidadas para representar cada uno de los subproblemas. Para mantener una consistencia con los fundamentos de la programación estructurada, los diagramas Nassi-Shneiderman no tienen representación para las instrucciones GOTO.
Los diagramas Nassi-Shneiderman se utilizan muy raramente en las tareas de programación formal. Su nivel de abstracción es muy cercano al código de la programación estructurada y ciertas modificaciones requieren que se redibuje todo el diagrama.
Los diagramas Nassi-Shneiderman son (la mayoría de las veces) isomórficos con los diagramas de flujo. Todo lo que se puede representar con un diagrama Nassi-Shneiderman se puede representar con un diagrama de flujo. Las únicas excepciones se dan en las instrucciones GOTO, break y continue.

Expresiones Condicionales.

En una expresión de relación tenemos uno o más valores operados por los operadores correspondientes. Consiste en consecuencia en dos valores operados entre sí, con un operador de relación. Los operadores son:

De relacion.

- *">" Mayor que*
- *"<" Menor que*
- *">=" Mayor o igual que*
- *"<=" Menor o igual que*
- *"=" Igual que*
- *"<>" Distinto que*

Estos operadores, también pueden operar con caracteres (por medio de código ASCII). Ej. :
"A" < "B" Tendría un valor lógico de Verdadero pues el valor ASCII de A es menor que el de B (El código ASCII esta ordenado de acuerdo con el abecedario pero en Inglés, es decir que letras como la Ñ o letras con signos de puntuación se consideran caracteres especiales y se encuentran en otro orden.)
Una expresión relacional obtiene un valor lógico (VERDADERO O FALSO).

Expresiones lógicas

- *NOT (No)*
- *AND (Y)*
- *OR (O)*

A	B	NOT A	A AND B	A OR B
V	V	F	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	F

Operador NOT: Operador que actúa sobre un solo valor. Este operador tiene la cualidad de invertir el valor lógico. Ej. :

NOT <Valor Lógico>
NOT VERDADERO ..... Tendría un valor FALSO
NOT FALSO..... Tendría una valor VERDADERO

Operador AND: La operación AND vale VERDADERO cuando los dos valores son verdaderos, de lo contrario vale FALSO. Ej. :

<Valor Lógico> AND <Valor Lógico>

Operador OR: Es similar a AND en el sentido de que llevará un valor lógico a izquierda y derecha, pero la diferencia estriba en que OR devuelve VERDADERO cuando cualquiera de los dos valores es VERDADERO.

Los operadores relacionales tienen prioridad sobre los lógicos, siempre y cuando ningún paréntesis indique lo contrario. La prioridad de los operadores lógicos es la siguiente:

1. NOT

2. AND
3. OR

Complejidad Algorítmica.

¿De qué hablamos cuando hablamos de complejidad? Resulta evidente que el tiempo real requerido por un computador para la ejecución de algoritmo es directamente proporcional al número de operaciones básicas que el computador debe realizar en su ejecución. Medir por lo tanto el tiempo real de ejecución equivale a medir el número de operaciones elementales realizadas. Desde ahora supondremos que todas las operaciones básicas se ejecutan en una unidad de tiempo. Por esta razón se suele llamar **tiempo de ejecución** no al tiempo real físico, sino al número de operaciones elementales realizadas. Otro de los factores importantes, en ocasiones decisivo, para comparar algoritmos es la cantidad de memoria del computador requerida para almacenar los datos durante el proceso. La cantidad de memoria utilizada durante el proceso se suele llamar **espacio** requerido por el algoritmo.
Al no ser única la manera de representar un algoritmo mediante un programa, y al no ser único el computador en el cual se ejecutará, resulta que la medida del tiempo será variable dependiendo fundamentalmente de los siguientes factores:

1) El lenguaje de programación elegido
2) El programa que representa
3) El computador que lo ejecuta

Por eso surge la necesidad de medir el tiempo requerido por un algoritmo independientemente de su representación y del computador que lo ejecute.

El análisis de algoritmos se encarga del estudio del tiempo y espacio requerido por un algoritmo para su ejecución. Ambos parámetros pueden ser estudiados con respecto al peor caso (también conocido como caso general) o respecto al caso probabilístico (o caso esperado).

En Ciencias de la Computación, el término **eficiencia algorítmica** es usado para describir aquellas propiedades de los algoritmos que están relacionadas con la cantidad de recursos utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza. La eficiencia algorítmica puede ser vista como análogo a la ingeniería de productividad de un proceso repetitivo o continuo.
Con el objetivo de lograr una eficiencia máxima se quiere minimizar el uso de recursos. Sin embargo, varias medidas (complejidad temporal, complejidad espacial) no pueden ser comparadas directamente, luego, cual de dos algoritmos es considerado más eficiente, depende de cual medida de eficiencia se está considerando como prioridad; por ejemplo la prioridad podría ser obtener la salida del algoritmo lo más rápido posible, o que minimice el uso de la memoria, o alguna otra medida particular.
Una diferencia significativa entre el **análisis de complejidad de algoritmos** y la **teoría de la complejidad computacional**, es que el primero se dedica a determinar la cantidad de recursos requeridos por un algoritmo en particular para resolver un problema, mientras que la segunda, analiza todos los posibles algoritmos que pudieran ser usados para resolver el mismo problema.

La importancia de la eficiencia con respecto a la complejidad temporal fue enfatizada por Ada Lovelace en 1843 como resultado de su trabajo con el motor analítico mecánico de Charles Babbage:
"En casi todo cómputo son posibles una gran variedad de configuraciones para la sucesión de un proceso, y varias consideraciones pueden influir en la selección de estas según el propósito de un motor de cálculos. Una objetivo esencial es escoger la configuración que tienda a minimizar el tiempo necesario para completar el cálculo."


Augusta Ada King, Condesa de Lovelace

Existen muchas maneras para medir la cantidad de recursos utilizados por un algoritmo: las dos medidas más comunes son la complejidad temporal y espacial; otras medidas a tener en cuenta podrían ser la velocidad de transmisión, uso temporal del disco duro, así como uso del mismo a largo plazo, consumo de energía, tiempo de respuesta ante los cambios externos, etc. Muchas de estas medidas dependen del tamaño de la entrada del algoritmo (Ej. la cantidad de datos a ser procesados); además podrían depender de la forma en que los datos están organizados (Ej. algoritmos de ordenación necesitan hacer muy poco en datos que ya están ordenados o que están ordenados de forma inversa).
En la práctica existen otros factores que pueden afectar la eficiencia de un algoritmo, tales como la necesidad de cierta precisión y/o veracidad. La forma en que un algoritmo es implementado también puede tener un efecto de peso en su eficiencia, muchos de los aspectos asociados a la implementación se vinculan a problemas de optimización.

La Teoría de la computabilidad:

Es la parte de la computación que estudia los problemas de decisión que pueden ser resueltos con un algoritmo o equivalentemente con la llamada máquina de Turing. La teoría de la computabilidad se interesa por cuatro preguntas:
¿Qué problemas puede resolver una máquina de Turing?
¿Qué otros formalismos equivalen a las máquinas de Turing?
¿Qué problemas requieren máquinas más poderosas?
¿Qué problemas requieren máquinas menos poderosas?
La teoría de la complejidad computacional clasifica las funciones computables según el uso que hacen de diversos recursos en diversos tipos de máquina.

Orden de complejidad.

Podemos decir que: **"el tiempo de ejecución del algoritmo es proporcional a una de las siguientes funciones y además cada jerarquía de orden superior tiene a las inferiores como subconjuntos."**

1- Algoritmos Polinomiales

N : tiempo de ejecución lineal, el tiempo es directamente proporcional a la cantidad de datos, ej. si N vale 100 tardara el doble que uno donde N vale 50.

logN : complejidad logarítmica, esto puede suceder en algoritmos con iteración o recursión no estructural, ejemplo: búsquedas binarias.

N-logN : complejidad cuasi-lineal, el tiempo de ejecución es N LogN. Si N se duplica, el tiempo de ejecución es ligeramente mayor del doble, un ejemplo es el quicksort.

$\mathbf{N^2}$: Complejidad cuadrática. Suele ser habitual cuando se tratan pares de elementos de datos, como por ejemplo un bucle anidado doble. Si N se duplica, el tiempo de ejecución aumenta cuatro veces.

$\mathbf{N^3}$: Complejidad cúbica. Como ejemplo se puede dar el de un bucle anidado triple. Si N se duplica, el tiempo de ejecución se multiplica por ocho.

$\mathbf{N^j}$: Complejidad polinómica (j>3). Al crecer la complejidad del programa es bastante mala.

2- Algoritmo Exponencial

$\mathbf{2^N}$: Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en sub- programas recursivos con dos o más llamadas interna

Ejecución de algoritmos y análisis asintótico.

Definición del Principio de Invarianza

Dos implementaciones distintas de un mismo algoritmo $t1_{(n)}$ y $t2_{(n)}$ no diferirán en su eficiencia en más de alguna constante multiplicativa

El **Principio de Invarianza** nos permite deducir que no existe una unidad que se deba utilizar para expresar la eficiencia teórica de un algoritmo.

En su lugar, expresaremos solamente el tiempo requerido por el algoritmo, salvo una constante multiplicativa.

Por este principio, todas las implementaciones de un mismo algoritmo tienen las mismas características, aunque la constante multiplicativa pueda cambiar de una implementación a otra.

¿Cómo medimos el tiempo de ejecución de un algoritmo?:

- Por medio de un elemento físico (ej. Reloj)
- Considerando las instrucciones a ejecutar y afectándolas por el tiempo de ejecución de cada una.

Ahora bien, como cada programa puede contener diferentes sentencias condicionales, esto quiere decir que la dependencia del tiempo estará dada por las condiciones que indiquen los datos ingresados. Por lo que entendemos que según esas condiciones, con alguna combinación de datos tendremos el tiempo mínimo de ejecución, Tmin (N) y por otro lado tendremos que otra combinación de datos dando por resultado el tiempo máximo de ejecución, Tmax (N)
De lo cual identificamos que:

- Tmin(N) es el mejor caso que podríamos obtener.
- Tmax (N) es el peor caso que obtendríamos.

Ahora bien, existirán una serie mayoritaria de casos que estarán entre ambos extremos, definiendo un valor promedio o más frecuente.
Tmin(N) <= T(N) <= Tmax (N)

Además de los parámetros anteriores, también podemos ver que existen algunos factores "externos", como ser:

- Calidad del código que genera el compilador.
- Velocidad de ejecución del procesador

Si consideramos que es muy sencillo el cambio de compilador y que prácticamente la velocidad de procesamiento se duplica año a año *(Ley de Moore, que todo el mundo predice su fin)*; debemos realizar nuestro análisis independizándonos de estos factores.
Otro factor es que en aquellos problemas chicos, el impacto de su tiempo de ejecución no es un factor crítico de los mismos.

El análisis de la eficiencia algorítmica nos lleva a estudiar el comportamiento de los algoritmos frente a condiciones extremas. Matemáticamente hablando, cuando **N tiende al infinito ∞,** es un comportamiento asintótico.

¿Cuál es Mejor?

La idea principal del análisis asintótico en informática es la de comparar funciones reales de variable natural *f:* **N**->**R**, para poder decir cuál tiene mejor comportamiento asintótico, es decir, cuál es menor cuando la variable independiente es suficientemente grande.
Si sabemos hacer esto con dos funciones, se podrá utilizar el resultado para comparar las funciones tiempo de los algoritmos y así determinar cuál de ellos tiene mejor comportamiento asintótico.
Relaciones de Dominación:

Si ***f*** y ***g*** son dos funciones de **N** en **R**, es decir *f:* **N**->**R** y *g:* **N**->**R** , se dice que ***g*** domina asintóticamente a ***f*** (o simplemente que ***g*** domina a ***f***) si existen $k \geq 0$ y $m \geq 0$ donde $k, m \in Z$ tales que para todo entero $n \geq m$ se verifica la desigualdad $|f(n)| \leq k|g(n)|$

Si ***g*** domina a ***f*** y $g(n) \neq 0$, entonces $\left|\frac{f(n)}{g(n)}\right| \leq k$ para casi todos los enteros ***n***, es decir, para todos, salvo una cantidad finita. En concreto, para todos los valores $n \geq m$ se verifica dicha desigualdad.

Teorema

La relación de dominación **<** definida por ***f < g*** sii ***g*** domina a ***f***, es una relación reflexiva y transitiva. Esto indica que dicha relación no es simétrica, es decir, si ***g*** domina a ***f***, no necesariamente ***f*** domina a ***g***.

f ≡ g sii ***f*** domina a *g* y *g* domina a *f*
[f] es la clase de equivalencia de ***f***

Analizando el Tiempo de Ejecución

Imaginemos problema de procesamiento de cálculo computacional científico, para el que se decide pedir a 4 equipos que diseñen e implementen un algoritmo de cálculo.
El primer equipo implementa un algoritmo de complejidad temporal **N**, el segundo $\mathbf{N^2}$,el tercero $\mathbf{N^3}$ y el cuarto $\mathbf{2^n}$.
Para evaluar los algoritmos se los somete a un N=10, 20, 30, 40, 50 y 60 (Tamaño de entradas)

Obteniendo el siguiente cuadro:

	10	20	30	40	50	60
N	0,00001	0,00002	0,00003	0,00004	0,00005	0,00006
N^2	0,0001	0,0004	0,0009	0,0016	0,0025	0,0036
N^3	0,001	0,008	0,027	0,064	0,125	0,216
2^n	0,001	1,0	17,90min	12,7dias	35,7años	366 sig

Saque sus propias conclusiones acerca de la problemática de la complejidad

Conceptos a tener en cuenta

• Un programa que se va a ejecutar muy pocas veces, hace que nuestro foco debe estar en la codificación y depuración del algoritmo, donde la complejidad no es el factor crítico a considerar.

• Un programa que se utilizara por mucho tiempo, seguramente será mantenido por varias personas en el curso de su vida útil, esto hace que los factores que debemos considerar

son los que se relacionan con su legibilidad, incluso si la mejora de ella impacta en la complejidad de los algoritmos empleados.

• Un programa que únicamente va a trabajar con datos pequeños (valores bajos de N), el orden de complejidad del algoritmo que usemos suele ser irrelevante, pudiendo llegar a ser incluso contraproducente.

• Un programa de baja complejidad en cuanto a tiempo de ejecución, suele demandar un alto consumo de memoria; y viceversa. Esto es una situación que debemos evaluar y entender como impactan ambos factores.

• Un programa orientado al cálculo numérico hace que debamos tener en cuenta más factores que su complejidad o incluso su tiempo de ejecución; en estos casos debemos considerar la precisión del cálculo, el máximo error introducido en cálculos intermedios, la estabilidad del algoritmo, etc.

Cálculo de complejidad

Loa algoritmos bien estructurados combinan las sentencias de alguna de las formas siguientes:

1. sentencias sencillas
2. secuencia (;)
3. decisión (if)
4. bucles
5. llamadas a procedimientos

Sentencias sencillas

Son las indicadas al principio de esta unidad, como ser, entre otras:

• Sentencias de asignación
• Entrada/salida

Las mismas, siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño esté relacionado con el tamaño N del problema.

La inmensa mayoría de las sentencias de un algoritmo requieren un tiempo constante de ejecución, siendo su complejidad O(1).

Secuencia

La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales.

Decisión (si)

La condición suele ser de O(1), complejidad a sumar con la peor posible, bien en la rama ENTONCES, o bien en la rama SINO

En decisiones múltiples (SINO SI, SELECCIONAR), se tomará la peor de las ramas.

Bucles

En los bucles con contador explícito, podemos distinguir dos casos:

• Que el tamaño N forme parte de los límites

• Que el tamaño N no forme parte de los límites.
Si el bucle se realiza un número fijo de veces, independiente de N, entonces la repetición sólo introduce una constante multiplicativa que puede absorberse.

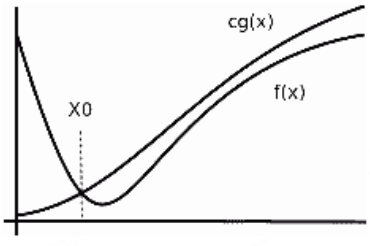
Llamadas a procedimientos
Su complejidad depende del contenido del procedimiento. Por ejemplo, el cálculo de la complejidad asociada a un procedimiento puede complicarse notablemente si se trata de procedimientos recursivos.
El costo de llamar no es sino una constante que podemos obviar inmediatamente dentro de nuestros análisis asintóticos.

Notación de Landeau

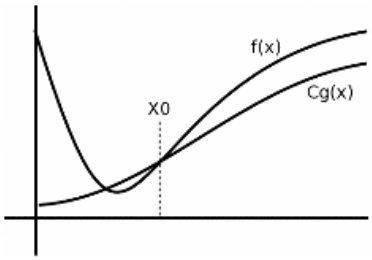

Edmund Georg Hermann Landau

En matemática, la Notación de **Landau**, también llamada "o minúscula" y "O mayúscula", es una notación para la comparación asintótica de funciones, lo que permite establecer la **cota inferior asintótica**, la **cota superior asintótica** y la **cota ajustada asintótica**.

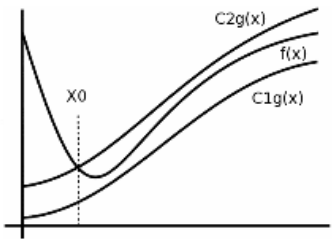
Una familia de funciones que comparten el mismo comportamiento asintótico será llamada un **Orden de complejidad.**



Cota superior asintótica



Cota inferior asintótica



Cota ajustada asintótica

• *Se adopta una notación especial llamada **O-grande** (big-Oh), por ejemplo **O(F(N))** para indicar que la cota superior del algoritmo es **F(N)**.*

• *En términos precisos, si **T(N)** representa el tiempo de ejecución de un algoritmo, y **F(N)** es alguna expresión para su cota superior, **T(N)** está en el conjunto **O(F(N))**, si existen dos constantes positivas* c *y* N_0 *tales que* $T(N) \leq c|F(N)|$ *para todo .*

• *El sólo saber que algo es **O(F(N)),** sólo nos dice que tan mal se pueden poner las cosas. Quizás la situación no es tan mala. De la definición podemos ver que si **T(N)** está en **O(N)**, también está en **O(N2)** y **O(N3)**, etc. (Por lo cual se trata en general de definir la mínima cota superior).*

La notación O, afirma que existe un punto N_0 tal que para todos los valores de N después de este punto, $T(N)$ está acotada por algún múltiplo de $F(N)$. Con el N lo suficientemente grande.

Por lo tanto, si el tiempo de ejecución $T(N)$ de un algoritmo es de $O(N^2)$, entonces **ignorando las constantes**, podemos garantizar que a partir de un punto se puede acotar el tiempo de ejecución mediante una función cuadrática.

Ejemplo: Consideremos el algoritmo de búsqueda secuencial para encontrar un valor especificado en un arreglo. Si el visitar y comparar contra un valor en el arreglo, requiere c_s pasos, entonces en el caso promedio $T(n) = c_s n/2$. Para todos los valores n>1 |cs n/2| ≤ cs|n|. Por lo tanto, por definición, T(n) está en O(n) para n0=1, y c=cs.

Existe una **notación** similar para indicar la mínima cantidad de recursos que un algoritmo necesita para alguna clase de entrada. La **cota inferior de un algoritmo**, denotada por el símbolo Ω, pronunciado "**Gran Omega**" u "**Omega**", tiene la siguiente definición:

• *T(n)* está en el conjunto *Ω(g(n))*, si existen dos constantes positivas *c* y *n0* tales que *|T(n)| ≥ c|g(n)|* para todo *n > n0*.

• Ejemplo: Si *T(n)=c1n2+c2n* para *c1* y *c2* > 0, entonces:
|c1n2+c2n| ≥ |c1 n2| ≥ c1|n2|
Por lo tanto, *T(n)* está en *Ω(n2)*.

Notación O grande (Big O)

Def. Sean f y g dos funciones no negativas sobre los enteros positivos.
Escribimos:

$$f(n)=O(g(n)) \text{ para toda } n \geq n_0$$

y decimos que $f(n)$ es de orden "a lo más" u orden
big O de $g(n)$ si existen constantes $c > o$ y n_o tales que:

$$f(n) \leq cg(n) \text{ para toda } n \geq n_0$$

La notación O, afirma que existe un punto N_0 tal que para todos los valores de N después de este punto, $T(N)$ está acotada por algún múltiplo de $F(N)$. Con el N lo suficientemente grande.
Por lo tanto, si el tiempo de ejecución $T(N)$ de un algoritmo es de $O(N^2)$, entonces **ignorando las constantes**, podemos garantizar que a partir de un punto se puede acotar el tiempo de ejecución mediante una función cuadrática.

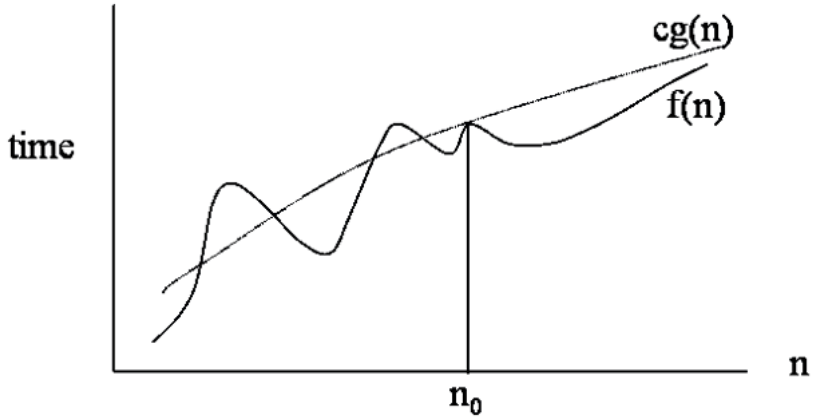
Ejemplo: Consideremos el algoritmo de búsqueda secuencial para encontrar un valor especificado en un arreglo. Si el visitar y comparar contra un valor en el arreglo, requiere c_s pasos, entonces en el caso promedio $T(n) = c_s n/2$. Para todos los valores n>1 |cs n/2| ≤ cs|n|. Por lo tanto, por definición, T(n) está en O(n) para n0=1, y c=cs.

- *Se adopta una notación especial llamada **O-grande** (big-O), por ejemplo **O(f(n))** para indicar que la cota superior del algoritmo es **f(n)**.*

- *En términos precisos, si **T(n)** representa el tiempo de ejecución de un algoritmo, y **f(n)** es alguna expresión para su cota superior, **T(n)** está en el conjunto **O(f(n))**, si existen dos constantes positivas c y n_0 tales que $T(n) \leq c|f(n)|$ para todo $n \geq n_0$.*

- *El sólo saber que algo es **O(f(n))**, sólo nos dice que tan mal se pueden poner las cosas. Quizás la situación no es tan mala. De la definición podemos ver que si **T(n)** está en **O(n)**, también está en **O(n2)** y **O(n3)**, etc. (Por lo cual se trata en general de definir la mínima cota superior).*

Decir que f(n) es de orden O(g(n)), significa:



Con otras palabras:

Def. Decimos que

$$f(n) \leq cg(n) \text{ para toda n} \geq n_0$$

Si y solo si,

$$0 \leq \lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right) = c$$

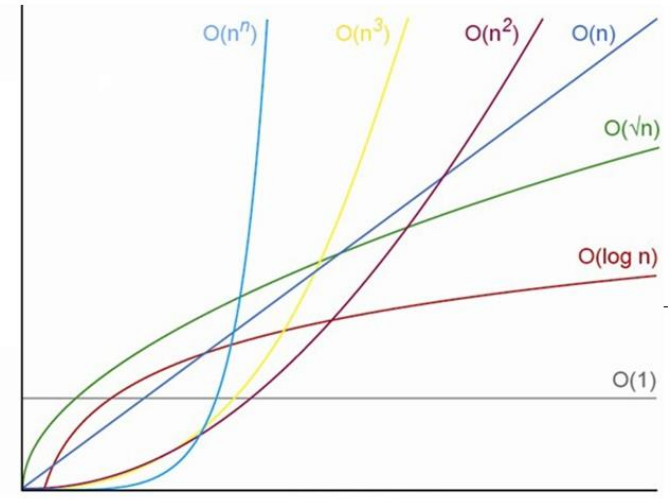
Ordenes de complejidad algorítmica usuales

O(**1**): Complejidad constante. Cuando las instrucciones se ejecutan una vez.

O(**log n**): Complejidad logarítmica. Esta suele aparecer en determinados algoritmos con iteración o recursión no estructural, ejemplo la búsqueda binaria.

O(**n**): Complejidad lineal. Es una complejidad buena y también muy usual. Aparece en la evaluación de bucles simples siempre que la complejidad de las instrucciones interiores sea constante.

O(**n log n**): Complejidad cuasi-lineal. Se encuentra en algoritmos de tipo divide y vencerás como por ejemplo en el método de ordenación quicksort y se considera una buena complejidad. Si n se duplica, el tiempo de ejecución es ligeramente mayor del doble.



O(**n^2**): Complejidad cuadrática. Aparece en bucles o ciclos doblemente anidados. Si n se duplica, el tiempo de ejecución aumenta cuatro veces. O(n^3): Complejidad cúbica. Suele darse en bucles con triple

anidación. Si n se duplica, el tiempo de ejecución se multiplica por ocho. Para un valor grande de n empieza a crecer dramáticamente.
O(n^p): Complejidad polinómica (a > 3). Si a crece, la complejidad del programa es bastante mala.
O(c^n): Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en subprogramas recursivos que contengan dos o más llamadas internas.

Clasificación

Clasificación por su Naturaleza

Los problemas de naturaleza algorítmica que no admiten solución por algoritmo son llamados no - computables.
Los problemas de decisión y no-computables son llamados de indecidíbles.
Los problemas para los cuales existen algoritmos de complejidad polinomial para resolverlos son llamados de tratables.
Los problemas que admiten solución y para los cuales comprobadamente no pueden ser resueltos por algoritmos de complejidad polinomial son rotulados de intratables.

Clasificación por el tipo de Respuesta

Problemas de Decisión: Su bbjetivo es responder SI o NO a una determinada indagación

Problemas de Localización: Su objetivo es encontrar, caso exista, una estructura que verifique las restricciones del problema, dicha estructura es denominada de solución viable .

Problemas de Optimización: Su objetivo es encontrar una estructura que verifique las restricciones del problema y optimice un criterio pre del problema y optimice un criterio pre-definido. Esto es, definido. Esto es, encontrar una solución viable que optimice un criterio pre encontrar una solución viable que optimice un criterio predeterminadoexponencial.

Clasificación por su Tratabilidad

La Clase P (Polynomial-time): Está constituida por todos los problemas comprobadamente tratables, esto es, problemas que pueden ser resueltos por algoritmos de complejidad polinomial. O dicho de otra forma, son problemas resolubles en tiempo polinómico con una máquina de Turing determinística.
P es conocido por contener muchos problemas naturales, incluyendo las versiones de decisión de programa lineal, cálculo del máximo común divisor, y encontrar una correspondencia máxima.
¿Se pueden resolver todos los problemas en tiempo polinomial?
– **No**. Existen muchos problemas que no se pueden resolver, no importa el tiempo involucrado

Ej:

- Resolución de Sistemas de Ecuaciones Lineales
- Contabilidad (registrar y/o modificar transacciones)
- Ordenar números, buscar palabras en un texto

- Juntar Archivos
- En general los sistemas operacionales (facturación, control de almacenes, planillas, ventas, etc.)
- Cualquier problema de la Programación Lineal
- Sistemas de transacciones bancarias
- En general los sistemas de información gerencial
- Programación de Tareas

La Clase NP (Non-Deterministic Polynomial-time): Está constituido por todos los problemas que pueden ser resueltos por algoritmos enumerativos, cuya búsqueda en el espacio de soluciones es realizada en un árbol con profundidad limitada por una función polinomial respecto al tamaño de la instancia del problema y con ancho eventualmente exponencial. O dicho de otra forma, son problemas resolubles en tiempo polinómico con una máquina de Turing no determinística.
La clase NP está compuesta por los problemas que tienen un certificado sucinto (también llamado testigo polinómico) para todas las instancias cuya respuesta es un SÍ. La única forma de que tengan un tiempo polinomial es realizando una etapa aleatoria, incluyendo el azar de alguna manera para elegir una posible solución, y entonces en etapas posteriores comprueba si esa solución es correcta.

Algunos Problemas

- Clique
- Camino Máximo (Dados dos vértices de un grafo encontrar el camino (simple) máximo),
- Ciclo Hamiltoniano(Ciclo simple que contiene cada vértice del grafo).
- Cobertura de Vértices y Aristas
- Coloración de Grafos
- Mochila Lineal y Cuadrática
- Optimización de Desperdicios
- Agente Viajero
- Gestión Optima de cortes
- Programación de Tareas

La Clase NP-Completa: Para abordar la pregunta de si P=NP, el concepto de la completitud de NP es muy útil. Informalmente, los problemas de NP-completos son los problemas más difíciles de NP, en el sentido de que son los más probables de no encontrarse en P. Los problemas de NP-completos son esos problemas NP-duros que están contenidos en NP, donde los problemas NP-duros son estos que cualquier problema en NP puede ser reducido a complejidad polinomial.

¿P=NP?

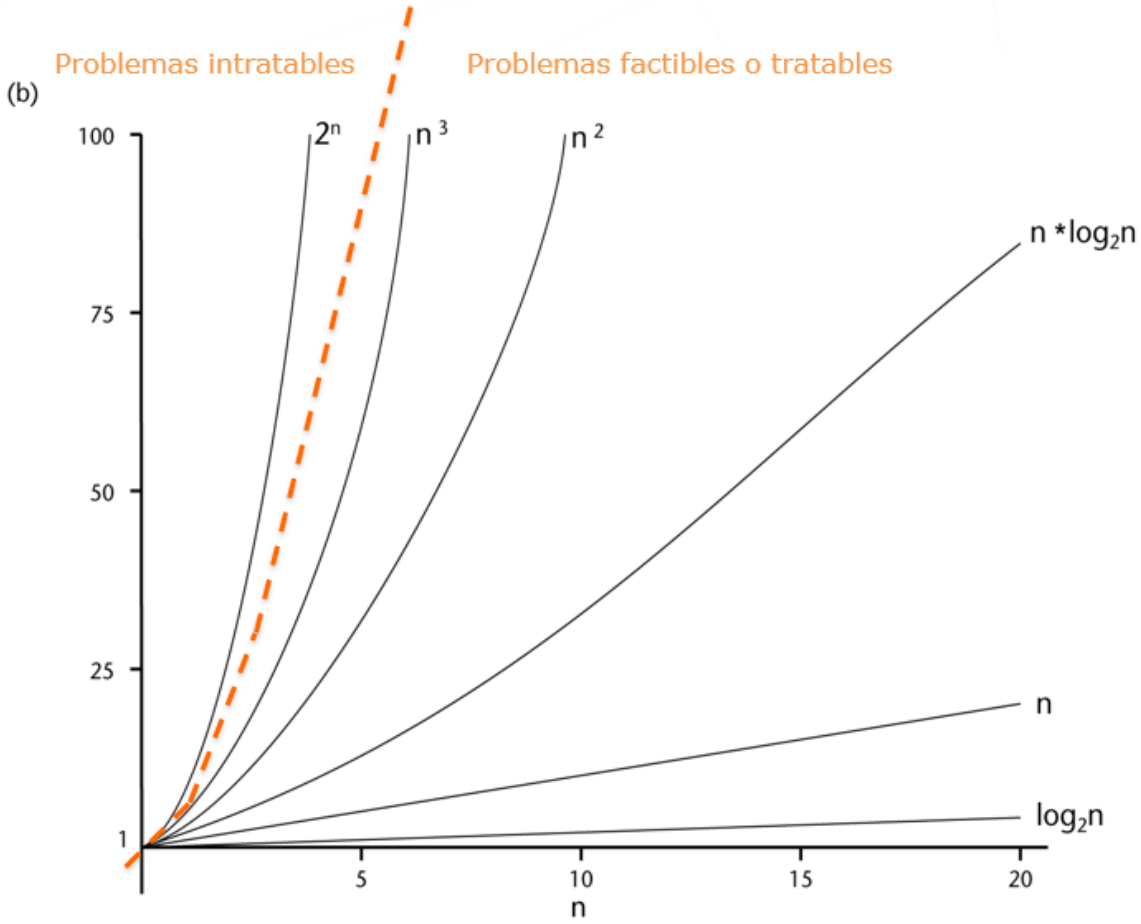
La cuestión de la inclusión estricta entre las clases de complejidad P y NP es uno de los problemas abiertos más importantes de las matemáticas. El Instituto Clay de Matemáticas (Cambridge, Massachusetts) (http://www.claymath.org/millennium-problems/p-vs-np-problem) premia con un millón de dólares a quién sea capaz de lograr la resolución de esta conjetura.

En algunos problemas, el comprobar la solución es más eficiente que calcularla. La complejidad de la función “elevar al cuadrado” es más simple que calcular la raíz cuadrada. ¿Qué tiene que ver todo esto con P=NP? Pues bien, P es la clase de complejidad que contiene problemas de decisión que se pueden resolver en un tiempo polinómico. P contiene a la mayoría de problemas naturales, algoritmos de programación lineal, funciones simples,... Por ejemplo la suma de dos números naturales se resuelven en tiempo polinómico (para ser más exactos es de orden 2n). Entre los problemas que se pueden resolver en tiempo polinómico nos encontramos con diversas variedades como los logarítmicos (log(n)), los lineales (n), los cuadráticos (n^2), los cúbicos (n^3),... Volviendo al ejemplo principal llegamos a la conclusión que la función de elevar al cuadrado está contenida en la clase P.
La clase de complejidad NP contiene problemas que no pueden resolverse en un tiempo polinómico. Cuando se dice que un algoritmo no puede obtener una solución a un problema en tiempo polinómico siempre se intenta buscar otro procedimiento que lo consiga mejorar. Frente a los problemas contenidos en P tienen métodos de resolución menos eficaces. Podemos ver que la operación de calcular la raíz cuadrada se encuentra contenida en esta clase.
Si nos resulta sencillo encontrar una solución para un determinado problema, sabemos comprobar si la solución es cierta (simplemente comparar), por lo que P es un subconjunto de la clase NP. La gran cuestión es si ocurre lo mismo a la inversa, es decir, si tengo un problema que sé comprobar fácilmente si un resultado es una solución, ¿sé calcular una solución sencillamente? ¿Todo problema se puede resolver en tiempo polinómico? Si alguien conoce la respuesta que se dirija al Instituto Clay y reclame su millón de dólares.

Problemas Tratables e intratables

Problemas Tratables: Son aquellos que se pueden resolver por un algoritmo en tiempo polinomial. La cota superior (upper bound) es polinomial.
Problemas Intratables: No se puede resolver por un algoritmo en tiempo polinomial. La cota inferior (lower bound) es exponencial.

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
n-log-n	$O(n \times \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(k^n)$, e.g. $O(2^n)$
factorial	$O(n!)$
super-exponential	e.g. $O(n^n)$



Ciertos problemas son tan difíciles que no existen algoritmos que los resuelvan. Un programa para el que no existirá nunca un algoritmo que lo resuelva es llamado ***irresoluble*** (*unsolvable*) o **no computable**. Un gran número de problemas son reconocidos como irresolubles.

Uno de los primeros problemas que se demostró son irresolubles es el *Halting problem o problema de Paro*: "Dado un programa arbitrario y un conjunto de entradas, ¿el programa eventualmente parará?"

Un número grande de problemas ***resolubles***, mantienen un estado indeterminado; se considera que son intratables, pero esto no se ha podido demostrar para ninguno de ellos.

La mayoría de estos problemas pertenecen a la clase de los **NP**-completos.

Muchos problemas prácticos se han identificado como **NP**-completos.

Un ejemplo de programa NP-Completo es el siguiente:
Dada una formula arbitraria de la lógica proposicional, verificar si existe una asignación de valores de verdad a sus variables, que la vuelva verdadera.
Este es el llamado *problema de satisfacibilidad*.

Problemas Decidibles: Son problemas computables y existe al menos un algoritmo capaz de resolverlos. Estos problemas a pesar de llevar mucho tiempo para su resolución son computables.
Dentro de esta categoría se encuentran los **Problemas Tratables** y los **Problemas Intratables**.

Problemas NO Decidibles: Son problemas que no son factibles obtener su solución. Aquí distinguimos dos subgrupos.

Problemas NO Computables: Un ejemplo de problema NO Computable lo constituye el famoso problema de Las Torres de Hanoi

Problemas Fuertemente No Computables: Un ejemplo de este lo constituye La Aritmética de PestBurger. Y cualquier algoritmo criptográfico como RSA, DES ...

Problemas Deterministas y No deterministas

Problemas Deterministas: En cualquier paso del algoritmo, sólo hay un siguiente paso posible.
Problemas No Deterministas: Puede existir más de un paso a elegir y no siempre se elegirá el mismo.

Propiedades de O grande (se pueden usar para simplificar)

1. Transitividad: Si *f(n)* está en *O(g(n))* y *g(n)* está en *O(h(n))*, entonces *f(n)* está en *O(h(n))*.
Esta regla nos dice que si alguna función *g(n)* es una cota superior para una función de costo, entonces cualquier cota superior para *g(n)*, también es una cota superior para la función de costo.
Nota: Hay una propiedad similar para la **notación** Ω y Θ**.**
2. Si *f(n)* está en *O(k g(n))* para cualquier constante *k>0*, entonces *f(n)* está *O(g(n))*
El significado de la regla es que se puede ignorar cualquier constante multiplicativa en las ecuaciones, cuando se use **notación** de **O-grande**.
3. Regla del valor máximo: Si *f1(n)* está en *O(g1(n))* y *f2(n)* está en *O(g2(n))*, entonces *f1(n) + f2(n)* está en *O(max(g1(n),g2(n)))*.
La regla expresa que dadas dos partes de un programa ejecutadas en secuencia, sólo se necesita considerar la parte más cara.
4. Si *f1(n)* está en *O(g1(n))* y *f2(n)* está en *O(g2(n))*, entonces *f1(n)f2(n)* está en *O(g1(n)g2(n))*.
Esta regla se emplea para simplificar ciclos simples en programas. Si alguna acción es repetida un cierto número de veces, y cada repetición tiene el mismo costo, entonces el costo total es el costo de la acción multiplicado por el número de veces que la acción tuvo lugar.
5. Igualdad de logaritmos: En la notación O la expresión

$$O(\log_a n) = O(\log_b n)$$

que, claramente es incorrecto en las matemáticas, pero es correcta en la notación O.

Esto se demuestra aplicando la propiedad de los logaritmos para el cambio de base $\log_a n = \frac{\log_b n}{\log_b a}$.
Notamos que el denominador es una constante y como en la notación O no se escriben las constantes queda demostrado.
6.Reflexiva: Es reflexiva dado que

$f(n) \epsilon O(f(n))$

La notación Ω, Gran Omega (Big Omega).

Def. Sean f y g dos funciones no negativas sobre los enteros positivos.
Escribimos:

$$f(n) = \Omega(g(n))$$

y decimos que $f(n)$ es de orden de "al menos" o orden *omega* de $g(n)$
si existen constantes $c > o$ y a tales que:

$$f(n) \geq c\ g(n) \quad \text{para toda} \quad n \geq a$$

Se adopta una notación especial llamada **Gran Omega** (Big- Ω), por ejemplo **Ω (f(n))** para indicar que la cota inferior del algoritmo es **f(n)**.

La notación Θ, Gran Theta (Big Theta).

Def. Sean f y g dos funciones no negativas sobre los enteros positivos.
Escribimos:

$$f(n) = \Theta(g(n))$$

y decimos que $f(n)$ es de orden de $g(n)$ o de orden *teta* de $g(n)$:

$$\text{si } f(n) = O(g(n)) \text{ y } f(n) = \Omega(g(n))$$

Se adopta una notación especial llamada Gran Theta (Big- Theta), por ejemplo Θ(f(n)) para indicar que la cota asintóticamente ajustada del algoritmo es f(n).

Notación o pequeña, o minúscula (Small o).

Def. Sean f y g dos funciones no negativas sobre los enteros positivos.
Escribimos:

$$f(n)=o(g(n)) \text{ para toda } n > n_0$$

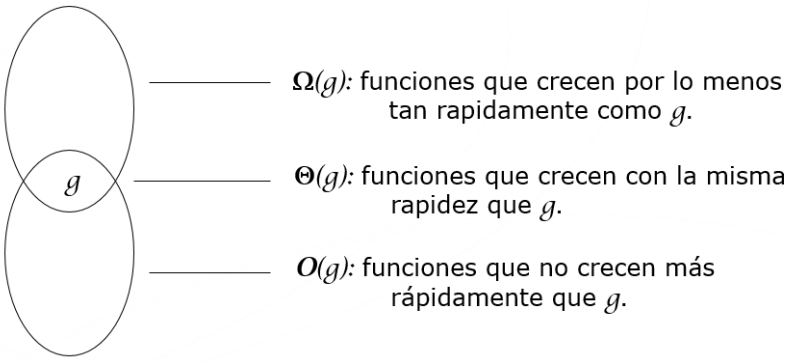
y decimos que $f(n)$ es de orden ***small o*** de $g(n)$
si existen constantes $c > 0$ y n_0 tales que:

$$f(n) < cg(n) \text{ para toda } n \geq n_0$$

Limitaciones del análisis asintótico

- **No es apropiado para pequeñas cantidades de datos.**
- **La constante implícita puede resultar demasiado grande en la práctica, independientemente de la tasa de crecimiento.**
  - Se da cuando un algoritmo es demasiado complejo
  - Se puede dar también porque nuestro análisis no diferencia los accesos de memoria de los accesos a disco. Nuestro análisis presupone memoria infinita.

Clasificación de funciones por su tasa de crecimiento asintótico



Reglas de cálculo

- ***Regla 1. Ciclos***
  El tiempo de ejecución de un ciclo es a lo más el tiempo de ejecución de las instrucciones que están en el interior del ciclo (incluyendo las condiciones) por el número de iteraciones.

- ***Regla 2. Ciclo anidados***
  Analizarlos de dentro hacia fuera. El tiempo de ejecución total de proposición dentro de un grupo de ciclos anidados es el tiempo de ejecución de la proposición multiplicado por los tamaños de entrada de todos los ciclos.

- ***Regla 3. Proposiciones consecutivas***
  Simplemente se suman, lo que significa que el máximo es el único que cuenta

- ***Regla 4. Condiciones***
  El tiempo de ejecución de una condición nunca es más grande que el tiempo de ejecución de la condición más en mayor de todos los tiempos de las proposiciones internas, ya sea cuando sea verdadera o falsa la condición.

Las reglas 1 y 2 parecen ser relativamente fáciles, y de hecho lo son hasta cierto punto, el problema se presenta cuando no se conoce a ciencia cierta la cantidad de iteraciones que tiene que ejecutar el algoritmo pueda llegar a su fin, en este caso primero hay que calcular primeramente la supuesta cantidad de iteraciones y luego aplicar las reglas 1 y 2.

Búsqueda Estática

Interna: Cuando los elementos entre los que buscamos están almacenados en memoria (array).

Externa: Cuando los datos entre los que buscamos están almacenados en un dispositivo externo; en este caso la búsqueda se realiza a partir de un determinado campo denominado campo clave.

- **Secuencial o lineal**
- **Binaria o dicotómica**
- **Por transformación de claves o HASHING**

Búsqueda secuencial y secuencial con centinela

Búsqueda secuencial : Sea A un vector con N elementos, para localizar el valor del elemento buscado dentro de A se comparará dicho valor con los diferentes miembros de A hasta encontrar el valor buscado o bien llegar hasta el final del vector, en el caso de que el elemento no se encuentre en dicho vector.

Búsqueda secuencial con centinela: Es una mejora del anterior en cuanto a que evita que tengamos que estar preguntando constantemente si hemos llegado al final del array.

El método consiste en asignar el valor buscado a la posición N+1 del array siendo N el número de elementos del array, cuando acabemos si LUG que es la variable que nos indica la posición del elemento es igual a N+a significará que el elemento no se encuentra en dicho array.
El inconveniente de este método es que para poder aplicarlo el array tiene que tener la capacidad suficiente para almacenar N+1 elementos.
Este método se llama centinela, porque al elemento que metemos en la posición N+1 se llama así.

Búsqueda binaria o dicotómica

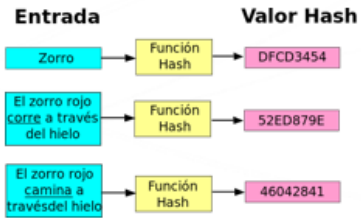
El problema de la búsqueda secuencial es que cuando el número de elementos es muy grande, el proceso de búsqueda se hace muy lento, por eso vamos a ver este método, que se basa en dividir el espacio de búsqueda o array en sucesivas mitades hasta encontrar el elemento buscado.
El inconveniente es que ***exige que el array esté ordenado*** para poder aplicarlo por ello el primer paso será ordenarlo.
La metodología es: se compara el elemento buscado con el de la mitad de la lista y si no es igual se decide en qué parte de la lista se seguirá buscando. Luego se vuelve a comparar con el valor medio de la sub-lista seleccionada (*y así sucesivamente*).
En el peor de los casos la complejidad sería aquella en la que el elemento no estuviera en la lista o estuviera en la última mitad a analizar, por tanto, como máximo haré tantas comparaciones como el número de mitades máximo en el que puedo dividir la lista.
Si N fuera el número máximo de elementos del array, K sería el número máximo de comparaciones sí
$2^k = N + 1$.
$K = \log_2 (N + 1)$
El caso más favorable sería aquel en el que sólo tuviera que hacer una comparación, porque el elemento buscado coincidiese con la primera mitad del array
$A + \log_2 (N + 1) = \log_2 (N + 1)$

Búsqueda por conversión de claves: HASHING

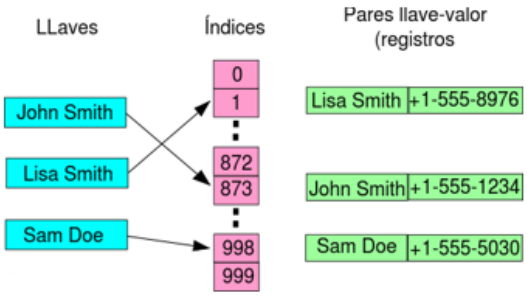
El término *hash* proviene, aparentemente, de la analogía con el significado estándar de dicha palabra en el mundo real: *picar y mezclar*. Donald Knuth cree que H. P. Luhn, empleado de IBM, fue el primero en utilizar el concepto en un memorándum fechado en enero de 1953. Su utilización masiva no fue hasta después de 10 años.
Un ejemplo práctico para ilustrar que es una tabla hash es el siguiente: Se necesita organizar los periódicos que llegan diariamente de tal forma que se puedan ubicar de forma rápida, entonces se hace de la siguiente forma - se hace una gran caja para guardar todos los periódicos (una tabla), y se divide en 31 contenedores (ahora es una "hash table" o tabla fragmentada), y la clave para guardar los periódicos es el día de publicación (índice). Cuando se requiere buscar un periódico se busca por el día que fue publicado y así se sabe en qué zócalo (bucket) está. Varios periódicos quedarán guardados en el mismo zócalo (es decir colisionan al ser almacenados), lo que implica buscar en la sub-lista que se guarda en cada zócalo. De esta forma se reduce el tamaño de las búsquedas de O(n) a - En el mejor de los casos O(1) y en el peor de los casos O(log(n)).

Para utilizar este método tampoco hace falta que el array esté ordenado pero si exige que cada elemento del array tenga asociada una clave.

Entonces, dado un array de N posiciones y dado un elemento a buscar en dicho array, se trata de convertir la clave de ese elemento a una de las N posiciones del array *(función hash)* y en esa posición será la que investigaremos si está o no el elemento.
Esta búsqueda exige la existencia de una función de direccionamiento o conversión de claves *(función hash)* que establezca la correspondencia entre claves y posiciones del array.
El inconveniente de este método es que si el rango de claves es muy grande no sería eficiente reservar espacio para todo el rango de claves (*a veces no sería ni posible*). Teniendo en cuenta que cada elemento puede tener cualquier clave del rango, si reservo menos posiciones de las que tiene ese rango puede darse el caso de que a más de un elemento la función de conversión le haga corresponder la misma posición en el array; esto se conoce como colisión. A cada una de las partes que colisionan se denominan SINÓNIMOS. La función de conversión tiene que ser tal que produzca el mínimo número de colisiones posibles.



Ejemplo de función Hash



Ejemplo de tabla Hash

Conclusión

Comprobar la eficiencia de un algoritmo equivale a medir la complejidad del tiempo asintótico en el peor caso **T(n)** de éste; o lo que es lo mismo, de qué orden es el número de operaciones elementales que requiere el algoritmo cuando el tamaño de la entrada de datos es suficientemente grande, en el peor caso.